Processes definition

Обычно ***программой*** называют совокупность файлов.
Это может быть набор исходных текстов, объектных файлов
Или, собственно, исполняемый файл.
Программа – это еще не процесс.

Для того чтобы программа могла быть запущена на выполнение,
система должна сначала создать *окружение* (***environment***)
или *среду* выполнения задачи, куда относятся
ресурсы памяти,
возможность доступа к устройствам ввода/вывода и
различным системным ресурсам, включая услуги ядра.

Это окружение (среда выполнения задачи) получило название ***процесса***.
Можно представить процесс как
совокупность данных ядра системы, необходимых для описания образа
программы в памяти и управления ее выполнением.

Можно также представить процесс, как программу в стадии ее
выполнения.

Процесс состоит из инструкций, выполняемых процессором,
текущих значений регистров и счетчиков команд,
текущих значений переменных,
данных и информации о выполняемой задаче,
такой как
размещение в памяти,
открытые файлы,
статус процесса и т.д..

Linux многозадачная система, поэтому одновременно исполняется
множество процессов,
каждый из которых следует **собственному** набору инструкций,
не передавая управление набору инструкций другого процесса.
Процесс считывает и записывает информацию
в ***свой*** *сегмент данных и в стек*,
но ему недоступны данные и стеки других процессов
(процессы ***изолированы*** друг от друга).

Несмотря на такую самостоятельность каждого процесса, процессам
зачастую необходимо как-то взаимодействовать с друг с другом.
(реализация даже простейшего ПАК или ПС не обходится обычно
одним единственным исполняющимся процессом).
То есть, изолированным процессам надо как-то
обмениваться данными (на runtime)

Для обмена данными и **синхронизации процессов** в Linux существует набор средств, образующих *систему межпроцессного взаимодействия (InterProcess Communication)*.

К этим средствам относятся :
каналы (pipes),
разделяемая память (shared memory),
семафоры (semaphores),
сигналы (signals),
очереди сообщений сообщения (message queue),
сокеты (sockets) и др.

Processes types & hierarchy

Системные процессы являются частью ядра и
всегда расположены в оперативной памяти.
Системные процессы запускаются особым образом при
инициализации ядра системы.
Выполняемые инструкции и данные этих процессов
находятся в ядре системы.
Они могут вызывать функции и обращаться к данным,
недоступным для остальных процессов.
Системными процессами являются, например:
shed (диспетчер свопинга),
vhand (диспетчер страничного замещения),
kmadaemon (диспетчер памяти ядра)
bdflush (диспетчер буферного кэша) и др.
К системным процессам следует отнести процесс ***init***, присутствующий в образе загрузки и
являющийся прародителем всех остальных процессов в Linux.
Хотя ***init*** не является частью ядра,
и его запуск происходит из исполняемого файла (/etc/init),
его работа жизненно важна для функционирования всей системы в целом.
Все процессы в системе Linux
принадлежат к единому дереву, начинающемуся с процесса ***init***.
В Linux реализована четкая иерархия процессов в системе.
На этапе запуска системы происходит предопределенная цепочка
запусков определенных процессов.
Задачей процесса i***nit*** является запуск всего остального нужным образом.

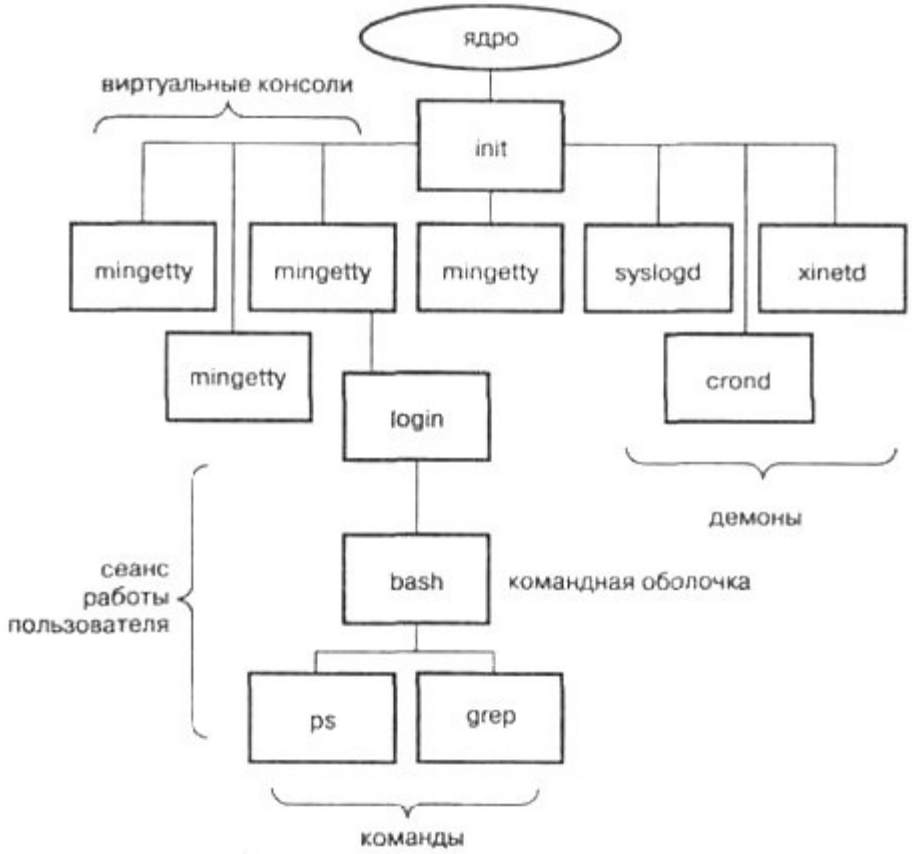
Init читает файл */etc/inittab*, в котором содержатся инструкции для дальнейшей работы.
Первой инструкцией, обычно, является запуск
скрипта инициализации /etc/init.d .

Здесь происходит проверка и монтирование файловых систем,
установка часов системного времени,
включение своп-раздела,

присвоение имени хоста и т.д.

Далее будет вызван следующий скрипт, который
переведет нас на «уровень запуска» по умолчанию.
Это подразумевает просто некоторый набор *демонов*,
которые должны быть запущены.
Демоны - это **неинтерактивные** процессы, которые запускаются
путем загрузки в память соответствующих им программ
(исполняемых файлов), и выполняются в *фоновом (background)* режиме.
Демоны обычно запускаются при инициализации системы
(но после инициализации ядра) и обеспечивают работу различных
подсистем (например,
системы печати, системы сетевого доступа и сетевых услуг,
системы терминального доступа, и т. п.)
Демоны **не связаны** ни с одним пользовательским сеансом работы и не
могут непосредственно управляться пользователем.
Большую часть времени демоны **ожидают**
пока тот или иной процесс запросит определенную услугу, например,
доступ к файловому архиву, печать документа и т.д.

Hierarchy shred



/etc/init.d/syslogd – скрипт, отвечающий за запуск и остановку
системного логгера (система журнальной регистрации событий **SYSLOG**,
записывает системные сообщения в файлы журналов */var/log*).

crond – демон читает пользовательские файлы *расписаний* из каталога */var/spool/cron/* и общесистемное расписание, хранящееся в файле */etc/crontab* и файлах, расположенных в каталоге */etc/cron.d/*.
После того, как расписания загружены в оперативную память, *crond* ежеминутно проверяет наличие записи в расписании, соответствующей текущему времени, и,
если такая найдена, запускает указанную команду .

Xined – демон Интернет-служб, управляет сервисами для Интернета. Демон прослушивает сокеты и если в каком-то из них есть сообщение определяет какому сервису принадлежит данный сокет и вызывает соответствующую программу для обработки запроса.

Последним важным действием *init* является
запуск некоторого количества *getty (get teletype),*
управляющих доступом к физическим и виртуальным терминалам (tty).
mingetty – виртуальные терминалы, назначением которых является слежение за консолями пользователей.
mingetty запускает программу *login*
– начало сеанса роботы пользователя в системе.

Процессы ждут, пока какой-нибудь пользователь не войдет в систему.
Задача *login*-а – регистрация пользователя в системе.
После успешной регистрации пользователя (пароль правильный) процесс входа в систему запускает оболочку, чаще всего грузиться командный интерпретатор пользователя (***shell***), например, ***bash*** (есть и другие) для обработки команд пользователя, команд которые, в свою очередь, могут запускать еще процессы.

Application Processes

К ***прикладным*** *процессам* относятся
все остальные процессы, выполняющиеся в системе.
Как правило, это процессы, порожденные
в рамках *пользовательского сеанса* работы.
Например, запуск команды ***ls*** породит соответствующий процесс.

Важнейшим пользовательским процессом является основной командный интерпретатор (*login bash)*, который обеспечивает
работу пользователя.
Он запускается сразу же после регистрации в системе,
а завершение работы *login shell* приводит к отключению от системы.
Пользовательские процессы могут выполняться
как в *интерактивном* (*foreground*),
так и в *фоновом (background)*
режимах, но в любом случае время их жизни (и выполнения) ограничено сеансом работы пользователя.
При выходе из системы все

пользовательские процессы будут **уничтожены**.
Интерактивные процессы **монопольно** владеют терминалом, и пока такой процесс не завершит свое выполнение, пользователь не сможет работать с другими приложениями.

Processes Attributes

Процесс имеет несколько атрибутов, позволяющих управлять его работой,
Каждый процесс имеет уникальный идентификатор (*Process ID*) ***PID***,
позволяющий ядру системы различать процессы.

Когда создается новый процесс,
ядро присваивает ему **следующий свободный**
(т. е. не ассоциированный ни с каким процессом) идентификатор.
Присвоение идентификаторов происходит по возрастающей,
т. е. идентификатор нового процесса больше,
чем идентификатор процесса, созданного перед ним.
Если идентификатор достиг максимального значения,
следующий процесс получит минимальный свободный *PID*
и цикл повторится.

Когда процесс завершает свою работу,
ядро освобождает занятый им идентификатор.
То есть **от запуска к запуску** один и тот же процесс
будет получать совершенно разные PID, в зависимости от наименьшего свободного номера в системе.

Атрибутом процесса является также идентификатор
родительского процесса (*Parent Process ID)* ***PPID*** ,
*то есть и*дентификатор процесса, породившего данный процесс.

Реальным идентификатором пользователя ***UID*** данного процесса
является идентификатор пользователя, запустившего этот процесс.
Эффективный идентификатор ***EUID*** служит для определения
прав доступа процесса к системным ресурсам
(в первую очередь к ресурсам файловой системы).

Обычно реальный и эффективный идентификаторы эквивалентны,
т. е. процесс имеет в системе те же права, что и пользователь,
запустивший его.

Однако существует возможность задать процессу более широкие права,
чем права пользователя путем установки флага *SUID*
*(*когда эффективному идентификатору присваивается значение
идентификатора владельца исполняемого файла, например, админ-а).
Аналогично идентификаторам пользователя с процессом ассоциируются
реальный и эффективный идентификаторы группы ***GID***, ***EGID*** .

Статический приоритет или *nice-приоритет* лежит в диапазоне от -20 до 19,
по умолчанию используется значение 0.
Значение –20 соответствует наиболее высокому приоритету,
nice-приоритет не изменяется планировщиком,
он наследуется от родителя или его указывает пользователь.

Динамический приоритет используется планировщиком для планирования выполнения процессов. Этот приоритет хранится в поле *prio* структуры t*ask_struct* процесса.
Динамический приоритет вычисляется исходя из значения
параметра *nice* для данной задачи путем вычисления
надбавки или штрафа, в зависимости от интерактивности задачи.
Пользователь имеет возможность изменять только статический приоритет процесса. При этом повышать приоритет может только *root*.
Существуют две команды управления приоритетом процессов:
nice и ***renice***.

Process status

Process table

Для управления процессами операционная система содержит **таблицу процессов**, в которой находится информация о состоянии процесса,
(счетчик команд, указатель стека, распределение памяти, состояние открытых

файлов), необходимая для переключений в различные состояния (готовности,
ожидания, исполнения и др.)

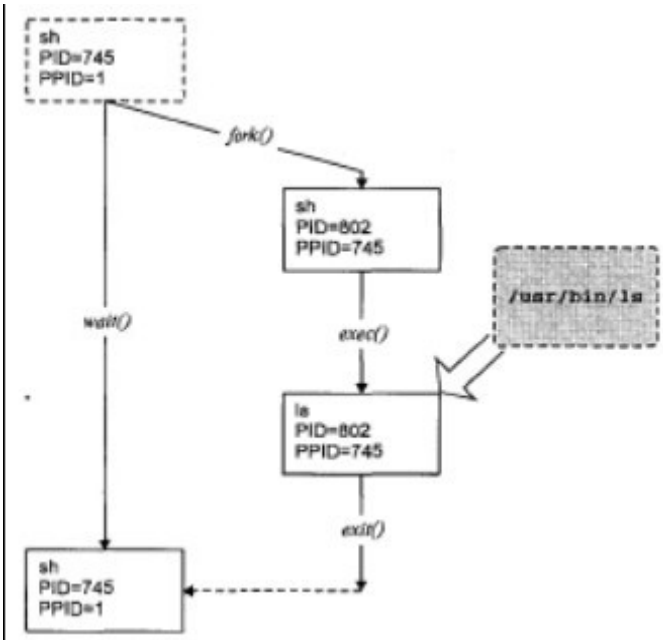
Примерное содержимое таблицы процессов:

Управление процессом	Управление памятью	Управление файлами
Регистры	Указатель на текстовый сегмент	Корневой каталог
Счетчик команд	Указатель на сегмент данных	Рабочий каталог
Слово состояния программы	Указатель на сегмент стека	Дескрипторы файла
Указатель стека		Идентификатор пользователя
Состояние процесса		Идентификатор группы
Приоритет		
Параметры планирования		
Идентификатор процесса		
Родительский процесс		
Группа процесса		
Сигналы		
Время начала процесса		
Использованное процессорнсе время		
Процессорное время дочернего процесса		
Время следующего аварийного сигнала		

Useful Commands

ps - Вывести список процессов
top - Интерактивно наблюдать за процессами
uptime - Посмотреть загрузку системы
w - Вывести список активных процессов для всех пользователей
free - Вывести объем свободной памяти
pstree - Отображает все запущенные процессы в виде иерархии

Process Start



Технология запуска и завершения процессов

Linux предоставляет два системных вызова *fork()* и *exec()*
для создания (порождения) процесса, и для запуска новой программы.
Фрагмент кода порождения процесса:

#include <sys/types.h>
#include <unistd.h>
. . .
pid_t fork(void);

Processes creation&termination inheritance

Порожденный или *дочерний* процесс, является точной копией *родительского* процесса. В частности, дочерний процесс наследует такие атрибуты родительского, как:
- идентификаторы пользователя и группы,
- переменные окружения,
- диспозицию сигналов и их обработчики,
- ограничения, накладываемые на процесс,
- все файловые дескрипторы,
- текущий и корневой каталог,
- управляющий терминал.

Виртуальная память дочернего процесса не отличается от образа родительского:
такие же сегменты кода (*code*), данных (*data*), стека (*stack*), разделяемой памяти (*shared memory*) и т. д.
После возврата из вызова *fork()* , который происходит и в родительский и в дочерний процессы, оба начинают выполнять одну и же инструкцию.

Немногочисленные различия между дочерним родительским процессами:
- дочернему процессу присваивается уникальный идентификатор
- идентификаторы родительского процесса PPID у этих процессов различны,
- значение, возвращаемое системным вызовом *fork()* различно для родителя и потомка.

При этом значение, возвращаемое родителю, равно *PID* дочернего процесса,
а дочернему процессу *fork() возвращ*ает значение, равное 0.
Если же *fork()* возвращает *-1*, то это свидетельствует о неудаче в выполнении, т.е. об ошибке
(естественно, возврат *-1* происходит только в тот процесс, который пытался выполнить системный вызов).

Таким образом, возвращаемое значение *fork()* позволяет определить, кто является родителем, а кто - потомком, и соответственно разделить дальнейшую функциональность этих процессов, которые будут выполняться параллельно..

Фрагмент кода программы:

```
. . .
int pid;
pid = fork();
if (pid == -1) {
        perror("fork"); exit (1);
}

If (pid == 0) {
/* Эта часть кода выполняется дочерним процессом */
                printf ("процесс-потомок\n");
}
Else            {
/* Эта часть кода выполняется родительским процессом */
printf ("процесс-родитель\n");
        }
. . .
```

Программа иллюстрирует процедуру клонирования процесса и *асинхронность поведения* процесса-родителя и процесса-потомка.

```
/* Программа forkdemo.cpp */
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
main()
{
int i;
   if (fork()) { /* Это процесс-родитель */
         for(i=0; i<1000 ; i++)
           printf("\t\tPARENT %d\n", i);
         }
         else { /* Это процесс-потомок */
           for(i=0; i<1000 ; i++)
                printf("CHILD %d\n",i);
            }
}
```

Асинхронность и конкурентность процессов в данном примере иллюстрируется случайным характером переключения вывода от процесса к процессу на этапе исполнения.

Невозможно точно предугадать порядок вывода, несмотря на то, что известен исходный текст программы.

Переключение ввода-вывода процессов зависит от длины кванта времени, выделяемого каждому из них планировщиком процессов, от буферизации, выполняемой библиотекой ввода-вывода, и даже от характеристик конкретного компьютера.

Processes creation&termination exec

Для загрузки другого исполняемого файла из процесса потомка предназначен системный вызов *exec().*

При выполнении *exec()* не создается новый процесс, а образ существующего (процесса-потомка) полностью заменяется на загружаемый из указанного при вызове *exec()* исполняемого файла.

Загружаемая программа наследует от процесса-потомка такие атрибуты как:
- идентификаторы процесса PID и PPID,
- идентификаторы пользователя и группы,
- ограничения, накладываемые на процесс,

- текущий и корневой каталоги,
- управляющий терминал,
- файловые дескрипторы, для которых не установлен
  флаг FD CLOEXEC.

Наследование характеристик процесса играет важную роль.
Так наследование идентификаторов владельцев процесса
гарантирует преемственность привилегий и, таким образом,
неизменность привилегий пользователя при работе в Linux.
Наследование файловых дескрипторов позволяет
установить направления ввода/вывода для нового процесса
или новой программы.

Системный вызов *exec()* представлен несколькими модификациями,
различающимися:

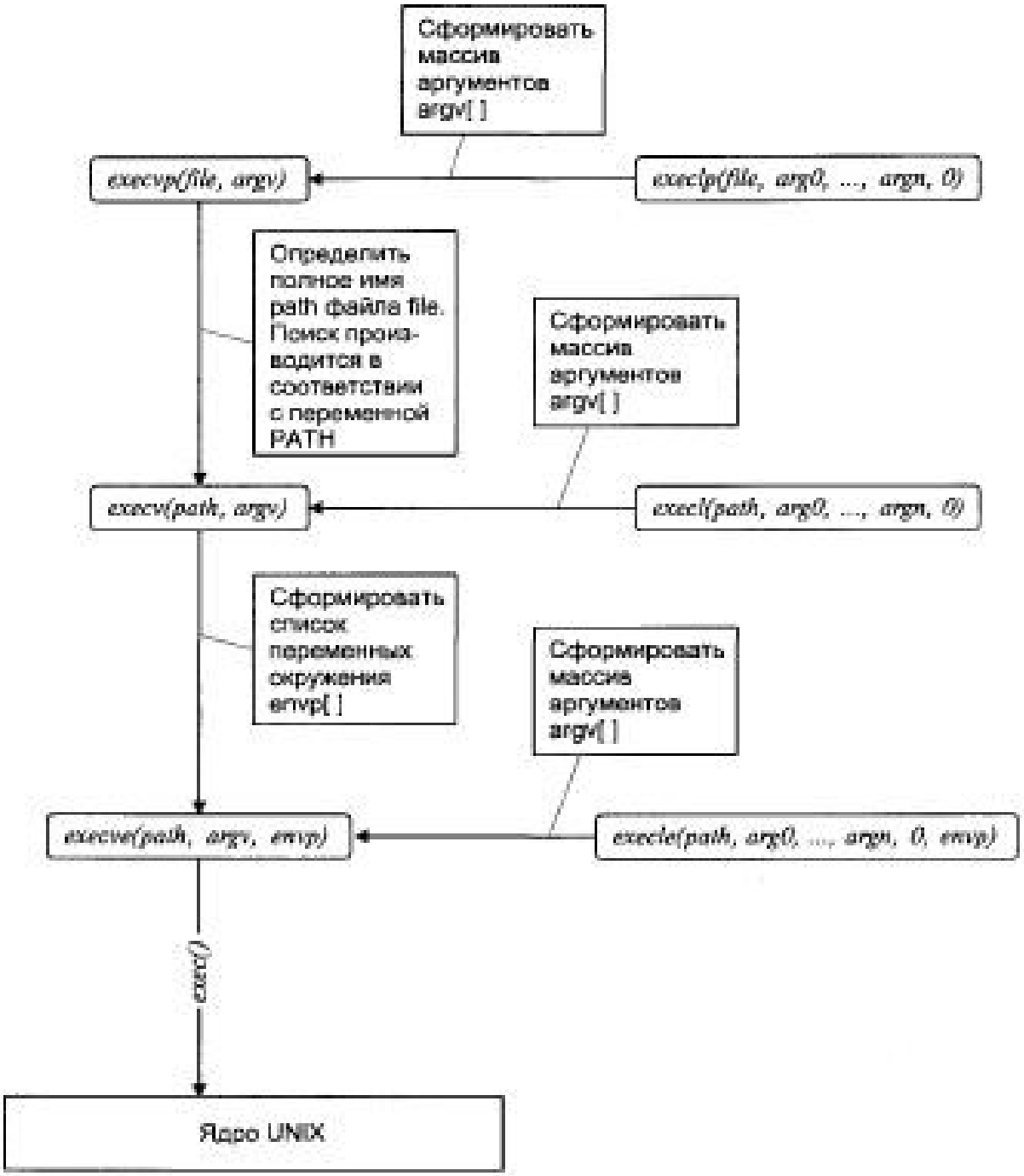
- тем, как задан путь к исполняемому файлу загружаемой программы
(абсолютно или относительно и используя переменную окружения PATH);

- тем, откуда берутся переменные окружения для загружаемой
программы
(сохраняются существующие или передается новый набор переменных);

- способом передачи аргументов командной строки
(в виде списка параметров при вызове (*explicit list*)
или как вектор (*vector*)).

Каждая разновидность вызова *exec()* имеет в мнемонике еще до двух
дополнительных символов.

Так, например, простейший в применении вызов *execlp()* ,
не указывает абсолютный путь, наследует окружение, и передает
параметры командной строки в виде списка. Имеет постфикс *lp*.
Такой же по семантике вызов, но передающий параметра
командной строки вектором, имеет постфикс *vp (т.е. это вызов execvp()).*
Аналог вызова *execlp()* , но передающий путь к загружаемому файлу в
абсолютном виде, имеет мнемонику *execl()*.
Аналог вызова *execvp()* , но передающий путь к загружаемому файлу в
абсолютном виде, имеет мнемонику *execv()*.
Наконец, вызовы *execle()* и *execve()* передают путь к загружаемому файлу
в абсолютном виде, но не наследуют переменные окружения,
а передают их новый набор, либо вектором, либо списком,
соответственно.
Картинка иллюстрирующая многообразие вызовов *exec()* :



Первый параметр вызова *exec()* всегда указывает имя загружаемого файла. Далее следует список аргументов *arg0, arg1, ... , argn* , либо указатель на вектор с этим списком.

Причем, требуется, чтобы первый элемент списка указывал на имя загружаемого файла (опять), а последний был нулевым указателем.

В программе *tinymenu* используется версия системного вызова *exec*, позволяющая передавать параметры командной строки списком, а переменные окружения наследовать.
С помощью вызова *execlp* в примере запускаются команды интерпретатора *shell*.

При этом, в нормальной ситуации, после вызова *execlp* и отработки соответствующей команды *shell*, управление, обратно, само по себе, не возвращается.

```
/* Программа tinymenu.cpp */
#include<stdio.h>
#include<unistd.h>
main()
{
        /* Фиксированный список команд */
static char *cmd[]={"who","ls","date"};
int i;
        /* Подсказка номера команды */
printf("0=who, 1=ls, 2=date:");
scanf("%d",&i);
        /* Запуск выбранной команды на исполнение */
   execlp(cmd[i], cmd[i], 0);
printf("Command not found\n");
        /* запуск не удачный */
}
```

Processes creation&termination wait

Как организовать возврат управления в родительский процесс из потомка, когда потомок завершается?

Посредством системного вызова *wait()* родительский процесс переводится в состояние ожидания, выход из которого происходит по событию завершения дочернего процесса, а именно, по выполнению в потомке вызова *exit()*.

```
/* Программа tinyexit.cpp */
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>
```

```
main()
{
        /* Фиксированный список команд */
static char *cmd[]={"who","ls","date"};
int i;
        while(1){
        /* Подсказка номера команды */
printf("0=who, 1=ls, 2=date:");
scanf("%d",&i);
        /* Если номер неверный, родительский процесс завершается */
        if(i<0 || I>2)
                exit();
        if (fork()==0){ /* Дочерний процесс */
        /* Процесс-потомок исполняет выбранную команду */
execlp(cmd[i], cmd[i], 0);
printf("Command not found\n");
        /* Запуск не удачный */
exit(1);
}
else
        {  /* Родительский процесс дожидается завершения дочернего */
        wait(0);
        }
   }
}
```

Системный вызов *wait()* возвращает значение *PID* идентификатора завершившегося дочернего процесса.
Параметр этого вызова позволяет передавать по ссылке информацию о *статусе завершения* процесса-потомка.

В программе *wait_parent* из процесса родителя запускаются три потомка, и затем системный вызов *wait()* отслеживает завершение каждого из них, сообщая, какой именно процесс закончил свое исполнение и какой *код завершения* был при этом передан.
В качестве процесса-потомка выступает программа *wait_child* , запускаемая из родителя с помощью *execlp* трижды
с разными параметрами командной строки.

В потомках случайным образом формируется *код завершения*
(для случаев нормального завершения),
а также варьируется и сам *способ завершения*.
При завершении по сигналу, номер сигнала передается процессу-родителю в младшем байте *статуса завершения*.

При нормальном же завершении, формируемый *код завершения*

находится в передаваемой информации о *статусе завершения*,
в байте, следующем после младшего (остальные байты обнуляются).

/* Программа wait_parent.cpp */

```
/* Процесс-родитель дожидается завершения процесса-потомка */
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>
main()
{
pid_t pid, w;
int i, status;
char value[3];
        for(i=0; i<3; ++i)
        { /* Запуск трех дочерних процессов */
        if ((pid=fork())==0){
                sprintf(value, "%d", i);
                execlp("wait_child", "wait_child", value, (char *)0);
                }
                else /* Подразумевается успешный запуск */
                        printf("Forked child %d\n", pid);
}
                        /* Ожидание завершения дочерних процессов */
while((w=wait(&status)) && w!=-1){
        if(w!=-1)
                printf("Wait on PID: %d returns status of:
                        %04X\n", w, status);
                }
                exit(0);
}
```

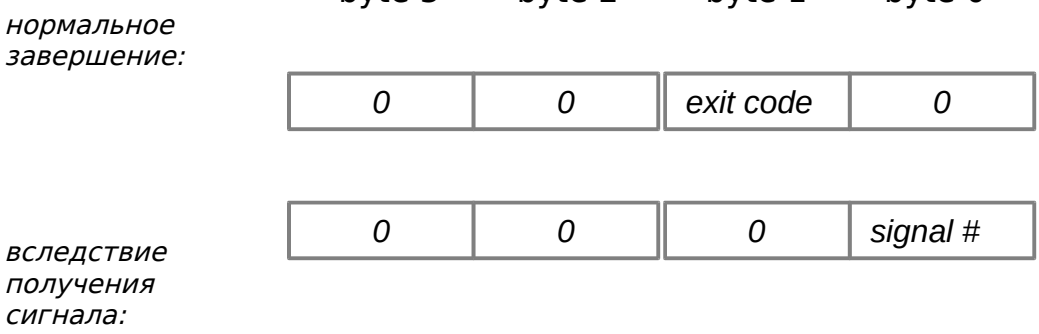
/* Программа wait_child.cpp */

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
#include<signal.h>
main(int argc, char *argv[])
{
```

```
pid_t  pid;
int  ret_value;
pid = getpid();
ret_value = (int) (pid % 256);
srand((unsigned) pid);
sleep(rand() %5);
if(atoi(*(argv+1)) % 2){
                /* Подразумевается, что при запуске argv[1] существует */
printf("Child %d is terminating with signal 0009\n", pid);
kill(pid, 9);           /* Процесс сам себя завершает - харакири */
}
else{
        printf("Child %d is terminating with exit (%04X)\n", ret_value);
        exit(ret_value);
        }
}
```

Байты статуса завершения

Status info:



Дополнительные возможности запуска процессов

Помимо основной группы системных вызовов *fork(), exec(), wait()* для запуска процессов в Linux могут быть использованы и другие возможности. К ним прибегают в случаях, если, например, существуют требования по производительности приложений или памяти системы.

Системный вызов

```
#include <sys/types.h>
#include <unistd.h>

pid_t vfork(void);
```

во многом повторяет функциональность рассмотренного *fork()* (потомок наследует дескрипторы файлов, диспозиции сигналов, текущий рабочий каталог) но, в отличие от него, блокирует родительский процесс при запуске потомка. Потомок разделяет память, включая стек, родителя, пока не завершится, и родитель не возобновит снова свое выполнение.

Системный вызов *clone() ,*во многом повторяя функциональность *fork() ,* создает дочерний процесс

Подробное описание этих системных вызовов доступно в системе помощи *man* , на практических занятиях эти вызовы могут быть использованы для целей сравнения с основными средствами запуска процессов в Linux, в качестве дополнительных заданий.